Permission-Based Ownership: Encapsulating State in Higher-Order Typed Languages

Neel Krishnaswami and Jonathan Aldrich
Carnegie Mellon University
{neelk+, aldrich+}@cs.cmu.edu

Abstract

Today's module systems do not effectively support information hiding in the presence of shared mutable objects, causing serious problems in the development and evolution of large software systems. Ownership types have been proposed as a solution to this problem, but current systems have ad-hoc access restrictions and are limited to Java-like languages.

In this paper, we describe System F_{own}, an extension of System F with references and ownership. Our design shows both how ownership fits into standard type theory and the encapsulation benefits it can provide in languages with first-class functions, abstract data types, and parametric polymorphism. By looking at ownership in the setting of System F, we were able to develop a design that is more principled and flexible than previous ownership type systems, while also providing stronger encapsulation guarantees.

Categories and Subject Descriptors D.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Theory, Verification

Keywords Ownership Types, Domains, System F, Type Theory, State, Modularity, Lambda Calculus, Permissions

1. Modularity and State

Modularity is at the core of software engineering. A well-designed module system allows developers to break a program into parts, but more importantly serves to hide design decisions made within each of those parts. This information hiding aspect of modules is the key criterion that supports separate understanding, separate development, and ease of change in large software systems [25].

Unfortunately, today's module systems do not effectively support information hiding in the presence of shared mutable state. The critical problem is aliasing: if a client gets an alias to a module's internal state, then information about how that state is represented and used within the module is no longer hidden. Thus the client may be affected by changes in the representation of the module, interfering with separate understanding, separate development, and software change. Furthermore, the client may violate representation invariants of the module, causing software defects or security holes.

```
class Class {
  private Object signers[];

  /* clients cannot call a method
   * returning an owned list */
  public Object [] getSigners() {
    return signers;
  }
}
```

Figure 1. In an early version of the JDK, the `Class.getSigners` method returned the internal list of signers rather than a copy, allowing untrusted clients to pose as trusted code.

1.1 Java Encapsulation Defect

For example, Figure 1 illustrates a security hole that was present in an early release of the Java development kit, version 1.1. In this defect, the security system function `Class.getSigners` returns a pointer to an internal array holding the principals that have signed a class. Clients can then modify the contents of the array, compromising the Java security model and potentially allowing malicious applets to pose as trusted code.

The key issue that led to the defect was that Java's module system is too weak to effectively encapsulate the array within the surrounding class object. Although the array was stored in a private field, it was all too easy for the developer to make a careless mistake that exposed the array to untrusted clients. Unfortunately, this problem is not unique to Java; other module systems may include more advanced mechanisms for hiding the visibility of members, but none provides a guarantee that aliases to those members do not escape to untrusted clients.

1.2 Ownership Types

Ownership types have been proposed as a practical mechanism for encapsulating internal, stateful objects within a surrounding *owner* object [22, 10, 8, 2, 9]. In an ownership type system, each type in the program is annotated with its owner object, and the type system ensures that only the owner can access the owned object. For example, in Figure 1 the `signers` array could have been marked **`owned`**, in which case the compiler would have flagged an error when the code

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PLDI'05, June 12–15, 2005, Chicago, Illinois, USA.
Copyright © 2005 ACM 1-59593-080-9/05/0006...$5.00.

returns the array to clients. Ownership is a stronger property than the visibility control provided by module systems, because it protects the owned object, not the variable that points to it.

In contrast to conventional module systems, ownership types provide a static guarantee that clients cannot refer to owned objects. Initial experience with ownership types has also shown that recent systems are flexible enough to apply to existing code with few changes [2].

1.3 Contributions

Despite the promise of ownership for encapsulating state in object-oriented systems, existing ownership type systems have a number of significant weaknesses, which we address in this paper.

Formal Models. The semantics of ownership has only been explored in the context of object calculi [11, 8] and models of Java [10, 9, 4, 6, 2, 1] which lack a direct connection to mathematical logic. In this paper, we introduce System F_{own}, a new formal model that naturally extends the type theory of System F [18] with references and ownership types.

In our system of permission-based ownership, an *ownership domain* represents a collection of references, functions, and existential packages. Each domain has a *kind* that expresses two forms of inter-domain access permission: the permission to create references, functions, and existential packages in another domain, and the permission to dereference references, call functions, and unpack packages in another domain. Our type system statically enforces these access permissions, ensuring the encapsulation of mutable state.

Advanced Language Features. Thus far, ownership types have only been integrated into Java-like languages that lack crucial advanced language features, such as first-class functions and ML-style abstract data types. Our formal model explains how ownership can benefit advanced object-oriented languages such as Scala [23] that include these features.

Expressiveness. Existing ownership systems rely on intuitive notions of encapsulation that are useful but overly conservative. In this paper, we show that guiding this intuition with type theoretic principles yields a system that is far more flexible, yet can enforce the stronger encapsulation guarantees than previous systems. For example:

- Our system can express challenging higher-order design patterns such as iterators and event callbacks in a cleaner way than previous systems, while ensuring that the state accessed via the iterator or callback is protected from clients.
- Our system uses abstract data types to allow the clients of a module to refer to private state while ensuring they cannot access it via pointer dereferences. Previous systems only support a weaker form of ADTs via inner classes.
- Our system's bounded existential quantification allows clients to use a domain to which they have access, even if they cannot name that domain, providing considerable practical flexibility compared to previous ownership systems.
- Our system allows a module to pass its private state to library functions, while ensuring that those library functions do not retain a persistent reference to the private state. Our system provides stronger encapsulation than previous work, in that it ensures that these library functions do not report information about the private state to clients through global variables.

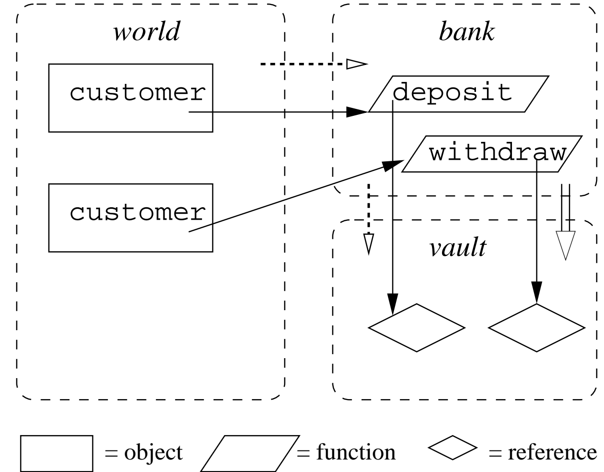


Figure 2. A conceptual view of ownership domains. The rounded, dashed rectangles represent ownership domains. Solid rectangles represent existential packages or objects, parallelograms represent functions, and diamonds represent reference cells. Dashed arrows represent access permissions, which allow arrows representing accesses, and double arrows represent permission to create entities in another domain.

1.4 Outline

The outline of the rest of this paper is as follows. In the next section, we describe Permission-Based Ownership, the ownership model on which our design is built. Section 3 introduces the syntax of System F_{own}, a formal model that integrates ownership seamlessly with type theory. We illustrate the expressiveness of System F_{own} through a series of examples. Section 4 gives the static and dynamic semantics for System F_{own} and states type soundness and encapsulation properties. Section 5 discusses related work, and Section 6 concludes.

2. Permission-Based Ownership

The purpose of Permission-Based Ownership is to give software engineers the ability to express and enforce high-level policies for encapsulation. The goal of our system is to encapsulate state; however, we believe that effectively encapsulating state also requires the encapsulation of functions and objects[1] that use that state. The intuition behind this goal is that the defect in Figure 1 would be the same if the array were instead represented as a List object or a pair of accessor functions. To fix the defect, we would need to encapsulate the List just as carefully as we would the corresponding array.

An encapsulation policy is a set of rules that control when a function can access or create a conceptually encapsulated entity such as another function, a reference cell, or an object. In this definition "access" means calling a function, dereferencing or assigning to a reference, or selecting a member of

[1] In this section we use the term "object" informally. In later sections, objects are represented by existential packages in the standard way.

an object. “Creation” means defining a new function, allocating a new reference cell, or creating a new object that is within an encapsulation boundary.

A fine-grained policy approach might attempt to specify access and creation permissions between every pair of entities in the system. However, such a policy would be prohibitively expensive to specify and enforce. Instead, we provide more high-level policy specifications that relate groups of functions, references and objects. These groups are called ownership domains.

2.1 Ownership Domains

An *ownership domain* represents a group of conceptually-related functions, reference cells, and objects (or more formally, existential packages). Ownership domains can be created at any time during program execution, and unlike regions in other systems, domains can outlive the scope in which they are declared. In our examples we will assume a domain $world$ that contains standard library functions, but there is nothing special about this domain.

Figure 2 illustrates the Permission-Based Ownership model used in System F_{own}. In the figure, domains are represented by dashed, rounded rectangles. Objects are shown as solid rectangles, functions are represented by parallelograms, and reference cells are represented as diamonds.

The somewhat anthropomorphic example in Figure 2 shows how ownership domains might be used to encapsulate state in a banking application. In this application, a number of customers interact with accessor functions in the banking interface, which in turn read and modify the data stored in the bank’s vaults. An important security constraint is that customers may not access the bank’s vaults directly, but instead must go through the interface functions.

We model this system using three ownership domains. The $world$ domain contains the `customer` objects. The *bank* domain includes the accessor functions for the bank, while the *vault* domain contains reference cells storing the amount of money in each account. Solid arrows show the calls made from the `customer` objects to the bank functions, as well as the accesses from the bank functions to the data in the vaults.

The `customer` object is part of the $world$ domain. It may call functions such as `deposit` and `withdraw` that are declared in the *bank* domain, which represents the interface to the bank. These functions in turn access state in the *vault* domain.

2.2 Policy Specifications

We want to ensure that the customer objects cannot directly access the data in the bank’s vaults. Permission-Based Ownership allows engineers to specify two kinds of permissions allowing one domain to affect another:

Access Permission. If one domain has access permission to another domain, then functions in the first domain can dereference or assign to references, call functions, or select a member of an object (i.e. unpack an existential) in the second domain. In Figure 2, access permission is represented by a single dashed arrow. Thus, functions that are part of the `customer` objects in the $world$ domain can call functions in the *bank* domain, and these functions can in turn access the state in the *vault* domain.

Access permission is *not* transitive in our system, ensuring that even though the $world$ domain has access permission to the *bank* domain and the latter has access to the *vault* domain, the $world$ domain does not have access to the *vault* domain. Thus, `customer` objects cannot access data in the vaults directly.

Creation Permission. If a `customer` object could create a function in the *bank* domain, it could simply have that function access the vaults on its behalf, bypassing the accessor functions provided by the bank. Therefore, we restrict the ability to create functions, reference cells, and objects in other domains using creation permissions.

In Figure 2, the *bank* domain has creation permission for the *vault* domain (shown with a double arrow), allowing the accessor functions to create reference cells for newly created bank accounts.

Permission Assignment. When a new domain is created, the creating entity assigns all of the access and creation permissions to and from that new domain. To ensure that domain creation cannot be used to circumvent access restrictions, our system ensures that the creating entity can only give the newly created domain the permissions that it itself possesses. For example, functions in the $world$ domain can create new domains that have either access or creation permission in the $world$ domain (since each domain has permission to access and create in itself) and access permission for the *bank* domain. These new domains, however, cannot be given any access to the *vault* domain, nor can they be given creation permission for the *bank* domain.

Static Checking. System F_{own} statically verifies that no inter-domain creations or accesses occur unless the source domain has the appropriate permission to create in or access entities in the target domain. Although our dynamic semantics preserve ownership information in order to easily prove a permission soundness theorem, ownership has no effect on the run-time semantics and can be erased during compilation.

3. System F_{own}

In this section, we present System F_{own}, an extension of System F [18] that incorporates references and permission-based ownership. Although we work in a functional context, many of our examples are inspired by object-oriented systems, where ownership types were first developed. For example, one of the well-known challenges for ownership types is iterators in a collection library, and we will show how our system handles this example.

Because we are working in the constructs of standard type theory, we do not explicitly model inheritance, but it can be encoded using existentials in the usual way. In the discussion below, we will use the terms object and existential package interchangeably.

3.1 Syntax

Figure 3 shows the source syntax of the System F_{own} language. System F_{own} is a formal core language inspired by Girard and Reynolds’ System F [18]. We take System F, including polymorphic functions and existential types, and add references and domain annotations.

Functions in the language are of the form $\lambda_\delta x{:}\tau.\ \ t$, where the annotation δ represents the ownership domain to which the function belongs. Reference expressions of the form $\text{ref}\ \ \delta\ \ t$ are annotated with the domain in which the reference is created. Modules are represented using existential packages, with an expression of the form

$$
\begin{array}{llll}
\text{Terms} & t & ::= & () \mid \lambda_\delta x{:}\tau.\ t \mid x \mid t_1\ t_2 \\
& & \mid & \text{ref}\ \delta\ t \mid !t \mid t_1 := t_2 \\
& & \mid & \text{pack}\ (\omega, t)\ \text{as}\ \exists_\delta x{:}K.\ \tau \\
& & \mid & \text{unpack}\ (\alpha, x) = t_1\ \text{in}\ t_2 \\
& & \mid & \Lambda_\delta \alpha{:}K.\ t \mid \Lambda \alpha{:}K.\ t \mid t[\omega] \\
& & \mid & \text{letdomain}\ z : \text{domain}(P)\ \text{in}\ t \\
& & \mid & \text{letdomain}\ z : \text{domain}(P)\ \text{into}\ t \\
\text{Types} & \tau & ::= & \text{unit} \mid \tau \rightarrow_\delta \tau' \mid \text{ref}\ \delta\ \tau \\
& & \mid & \forall_\delta \alpha{:}K.\ \tau \mid \exists_\delta \alpha{:}K.\ \tau \\
& & \mid & \forall \alpha{:}\text{domain}(P).\ \tau \mid \alpha \\
\text{Domain} & \gamma, \delta & ::= & \alpha, \beta, \ldots \\
& \omega & ::= & \tau \mid \delta \\
\text{Kinds} & K & ::= & \text{domain}(P) \mid \text{type} \\
\text{Permissions} & P & ::= & P, _ \Rightarrow \delta \mid P, \delta \Rightarrow _ \mid P, _ \Rightarrow _ \\
& & \mid & P, _ \rightarrow \delta \mid P, \delta \rightarrow _ \mid P, _ \rightarrow _ \\
& & \mid & \epsilon
\end{array}
$$

Figure 3. System F_{own} Source Syntax

"pack (ω, t) as $\exists_\delta \alpha{:}K.\ \tau$". Again, they have a domain annotation.

System F_{own} supports explicit parametric polymorphism with an expression of the form $\Lambda_\delta \alpha{:}K.\ t$. These type lambdas are as usual annotated with a domain, in which their body is typechecked, and can quantify over both types and domains.

For each type binding, a kind is given. Our system has two families of kinds. The first is the kind of types, which we use for conventional parametric polymorphism. We also have another family of kinds for domains, with a domain kind for each possible set of access permissions. Permissions include access permission ($\rightarrow$) and creation permission ($\Rightarrow$) to and from a domain ($\delta \rightarrow _$ vs. $_ \rightarrow \delta$). Using kinds to treat types and domains in a unified way allows universal and existential quantification to be used over both.

Normally, functions (and type functions) can only be created in a domain to which the current entity has creation permission. This restriction, however, prevents us from writing library functions that can act on arbitrary client domains. Therefore, System F_{own} provides a special form of domain polymorphism $\Lambda\alpha{:}\text{domain}(P).\ t$ that can be invoked from (nearly) any domain, but whose body is typechecked in the argument domain α. We ensure that this form cannot be abused by restricting P to contain only access and creation permission for the domain α (i.e., P may only contain $_ \rightarrow _$ and $_ \Rightarrow _$).

Domains are created using the letdomain constructs. An expression "letdomain $\alpha : K$ into t" or "letdomain $\alpha : K$ in t" creates a new domain with the permissions specified in K, and binds it to the variable α. The difference between the "into" and "in" forms is whether the body t is typechecked in the newly-created domain, or in the current domain. In order to avoid privilege escalation, we statically check letdomain forms to ensure that the newly-created domains cannot be given any permissions that its creator does not have.

Following Mitchell and Plotkin, System F_{own} uses existential quantification to model type abstraction [20]. In addition

$$
\begin{array}{l}
\text{val}\ \mathit{increment} = \Lambda\ d : \text{domain}(_ \rightarrow _)\ . \\
\quad \lambda_d\ x : \text{ref}\ d\ \mathit{int}\ .\ x := (!x) + 1;\ !x \\
\\
\text{val}\ \mathit{COUNTER} = \\
\ \ \text{letdomain}\ \mathit{pub} : \text{domain}(\mathit{world} \rightarrow _, _ \rightarrow _, _ \Rightarrow _)\ \text{into} \\
\ \ \text{letdomain}\ \mathit{owned} : \text{domain}(\mathit{pub} \Rightarrow _, \mathit{pub} \rightarrow _, _ \rightarrow _)\ \text{in} \\
\ \ \text{pack}\ (\mathit{pub}, \\
\quad \text{pack}\ (\text{ref}\ \mathit{owned}\ \mathit{int}, \\
\qquad \{\ \mathit{init}\ =\ \text{ref}\ \mathit{pub}\ 0, \\
\qquad \mathit{create}\ =\ \lambda_{pub}\ x : \text{unit}\ .\ \text{ref}\ \mathit{owned}\ (!\mathit{init}), \\
\qquad \mathit{inc}\ =\ \lambda_{pub}\ x : \text{ref}\ \mathit{owned}\ \mathit{int}\ .\ \mathit{increment}\ \mathit{owned}\ x \} \\
\quad)\ \text{as} \ldots \\
\ \ )\ \text{as}\ \exists_{world}\ \mathit{public} : \text{domain}(\mathit{world} \rightarrow _, _ \rightarrow _, _ \Rightarrow _)\ . \\
\quad \exists_{world}\ t : \text{type}\ . \\
\quad \{\ \mathit{init}\ :\ \text{ref}\ \mathit{public}\ \mathit{int}, \\
\quad\ \ \mathit{create}\ :\ \text{unit} \rightarrow_{public} t, \\
\quad\ \ \mathit{inc}\ :\ t \rightarrow_{public} \mathit{int}\ \}
\end{array}
$$

Figure 4. A counter abstract data type in System F_{own}.

to the familiar case of existential types, System F_{own} permits us to write expressions with existential domains. We make use of this feature to export domains out of the lexical scope of a `letdomain` expression.

Types τ include arrow types, reference types, universal types, and existentials. Just as function, reference and polymorphic and existential terms are annotated with domains, so too are their types. Arrows, reference types, and universal and existential types are all annotated with the domain to which they belong. The sole exception to this is the special universal type corresponding to the special domain polymorphism term; a term of this type must run in any domain, and so we don't need to specially annotate it with a home domain.

Like System F (and unlike core ML), System F_{own} supports impredicative polymorphism; that is, type variables can range over polymorphic types. This means we can model simple forms of ML-style functors and first class modules with functions that return existential packages. The most sophisticated uses of functors, which rely on full dependent sums, do not fit into this model.

3.2 Expressiveness

We demonstrate the expressiveness of System F_{own} through a series of examples. Some of our examples assume records, case statements, etc., which can be encoded in the standard way.

Counter. Figure 4 shows a *COUNTER* module defining a counter abstract data type. This module illustrates how ownership domains can be used to encapsulate internal state even in the presence of abstract types, state-manipulating library functions, and references that are shared across module boundaries.

The counter module defines a *pub* domain that will hold its public functions and reference cells, and an *owned* domain that will hold the reference cells used to implement the counters. The permissions for the *pub* domain, for example, give the *world* domain access permission to *pub* ($\mathit{world} \rightarrow _$), and give *pub* create and access permission to itself ($_ \rightarrow _$ and $_ \Rightarrow _$). The module defines an abstract type t for the counter, implemented as a reference cell in the *owned* domain.

```
rec val map : ∀ d : domain(_ → _) . ∀_d a, b : type .
      (a →_d b) →_d (a list →_d b list)
   = Λ d : domain(_ → _) .
      Λ_d a : type .
      Λ_d b : type .
      λ_d f : a →_d b .
      λ_d l : a list .
      case l of nil => nil
         | x::xs => (f x) :: map d a b f xs
```

Figure 5. A polymorphic map function in System F_{own}.

When a client wants to create a counter, they will first store the initial value of the counter into the *init* reference cell, then call *create* to get a new counter. The implementation of *create* simply creates a fresh reference initialized to the value stored in *init*.

The counter can be incremented by calling the *inc* function. The implementation of *inc* calls the *increment* library function, which is polymorphic in the domain d of the reference cells it increments.

Discussion. Although simple, this example illustrates several ways in which our design is more flexible yet provides more encapsulation than any previous ownership type system. On the flexibility side, we are able to implement the abstract type t in terms of a reference, allowing clients to refer directly to that reference, but ensuring that they cannot dereference that reference, since its type is held abstract and in any case, its domain is inaccessible. No previous ownership type system supports abstract types of this form, although inner classes can provide some of the expressiveness [8, 6].

On the encapsulation side, like other ownership type systems, our system provides a clear guarantee that no client can access the hidden references directly. Simply observing that the counter references are annotated with the *owned* domain, and that the *world* domain does not have access to the *owned* domain, is sufficient to guarantee this. If the implementor of the *COUNTER* module made a mistake that would expose an internal reference—for example, just using the *init* reference when creating a counter instead of creating a new reference initialized to the same value—the type system would notice the inconsistent domain annotations and would flag the error.

Our type system goes further than previous ownership systems, however, in that it allows developers to pass references to domain-polymorphic functions without the danger that those functions might break the module's abstraction guarantee. The *increment* function is defined with the special domain polymorphism form, so that the body of the function is typechecked as if it were within the argument domain. This gives *increment* permission to dereference and assign to the references passed to it.

However, because the body of *increment* is typechecked as if it were in a domain that has only permission to access or create in itself, the body of the function loses access to any other domains that were previously in scope. For example, the *increment* function would not be able to access state in the *world* domain. This ensures that although *increment* has access to the internal state of the *COUNTER* module, it cannot expose that state to clients, or even report any information about the state to clients through global references.

Our type system's support for writing domain polymorphic functions that do not violate abstraction is essential not only for using library code safely, but also for many other common programming idioms including callback functions and plugin modules. In comparison, all previous ownership type systems either prohibit fully polymorphic functions like *increment*, or else allow such functions to break the abstraction of *COUNTER* by storing the reference in an object [2], or reporting information about the references to clients through global state [8, 6].

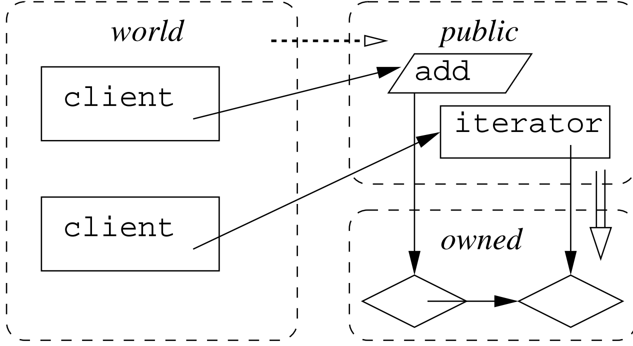
Comparison to Conventional Systems.
In this simple example, it is easy to see that the *counter* reference is never accessed outside functions in the *COUNTER* module and domain polymorphic functions like *increment*. However, the JDK bug described earlier shows that checking encapsulation properties like these is nontrivial in larger systems. Checking this property in a conventional module system would be considerably more difficult. For example, checking to make sure that the *COUNTER* interface contains no references is insufficient, because the interface does have such a reference—one must also check that the references used for counters don't alias the *init* reference. We cannot use an effect system to ensure that clients don't access the reference because in fact we want to allow them to do so–as long as they access it through the interface of *COUNTER*. Even doing an escape analysis is insufficient, because the *counter* reference actually escapes twice: to clients through the abstract type t, and to the *increment* library.

In other module systems, it would be necessary to manually verify that clients cannot use some function in *COUNTER* to convert a t into the underlying reference, and also to check that *increment* does not expose the reference to clients. These checks are automatically guaranteed by our type system. As mentioned above, since clients cannot access the *owned* domain inside *COUNTER*, there is no way they can possibly read or write the underlying reference.

Map. Figure 5 shows a *map* function that is polymorphic in the argument type, return type, and domain of the function to be mapped. As with the *increment* function above, System F_{own}'s type system restricts the direct side-effects of *map* to calling functions and accessing references in domain d (in this case, of course, *map* calls the function f). This example demonstrates that System F_{own} can express an important functional programming idiom, but also shows how System F_{own}'s domains help in reasoning about the effects of a function – we can tell from the type that *map* cannot have any side-effects except those that arise through calling its argument function f.

Note that unlike an effect system, we restrict only the direct effects of *map*, not the transitive side-effects that could occur through the call to f. This feature makes our system more lightweight in practice compared to effect systems, which must track the transitive effects of each function.

Object-Oriented Sequence ADT. Although presented in the context of System F, our type system can also express typical object-oriented idioms that present challenges to many previous ownership type systems. For example, Figure 6 shows a polymorphic sequence abstract data type, *SEQ*. The ADT is implemented in a typical object-oriented style, with mutator functions like *add* and support for iterating over the sequence with an iterator object. The diagram shows the in-




```
val ITER = Λ iter : domain(_ ⇒ _) .
      Λ_iter owned : domain(iter ⇒ _, iter → _) .
      Λ_iter t : type .
      λ_iter head : t list .
   let cur = ref owned head in
   { hasNext = λ_iter x : unit . isCons (!cur)
     next = λ_iter x : unit .
               let v = hd(!cur) in cur := tl(!cur); v }

val SEQ = Λ_world t : type .
   letdomain public : domain(world → _, _ ⇒ _, _ → _) into
   letdomain owned : domain(public ⇒ _, public → _) in
   let head = ref owned nil in
   pack (public,
      { add = λ_public x : t . head := cons x (!head),
        iter = λ_public x : unit . ITER public owned t (!head) }
   ) as ∃_world public : domain(world → _) .
      { add : t →_public unit,
        iter : unit →_public
                  { hasNext : unit →_public bool,
                    next : unit →_public t }
      }
```

Figure 6. The SEQ module implements a polymorphic sequence abstract data type. Iterators over the sequence are defined by the *ITER* module.

tended design of the system, which is similar to the bank example given earlier: the client can access the sequence and its iterators, and the iterators can access the underlying linked list, but the client should not be able to get to the underlying list directly.

The *SEQ* module is parametric in the type t of elements in the sequence. Like the *COUNTER* module above, the *SEQ* module defines a $public$ domain accessible from $world$, and a $owned$ domain for the private state in the sequence. That state is represented by a reference cell in the $owned$ domain which points to the head of the list.

The module itself is implemented as a package that exposes the $public$ domain as well as two functions in that domain. The add function mutates the sequence by adding another element to the front of the sequence. The $iter$ function returns an iterator over the sequence, using the *ITER* module defined above.

The iterator is implemented with the special domain-polymorphism form, just like the $increment$ function in the previous example, thus ensuring that the iterator implementation will not break the abstraction boundary of the sequence. The nested polymorphic function illustrates bounded domain polymorphism: the $owned$ domain passed in must allow the $iter$ domain to create and access state in

Values	v	::=	$\underline{() \mid l} \mid \overline{\lambda}_m x{:}\tau.\ t$
		|	$\underline{\text{pack}}\ (\omega, v)$ as $\exists_m \alpha{:}K.\ \tau$
		|	$\overline{\Lambda}_m \alpha{:}K.\ \tau \mid \overline{\Lambda}\alpha$:domain(). τ
Expr	t	::=	$t_s \mid v \mid t@m$
Store	μ	::=	$\epsilon \mid \mu, l : v$
Store typ.	Σ	::=	$\epsilon \mid \Sigma, l : \tau$
Environment	Γ	::=	$\epsilon \mid \Gamma, x : \tau \mid \Gamma, \alpha : K$
Contexts	C	::=	$\Box \mid C\ t \mid v\ C \mid \ \mid C\ [\omega] \mid$ ref δ C
		|	pack (ω, C) as $\exists_\delta \alpha{:}K.\ \tau$
		|	unpack $(\alpha, x) = C$ in t

Figure 7. System F_{own} Intermediate Forms and Values

it. When the iterator is used, the $iter$ and $owned$ domains are instantiated with the $public$ and $owned$ domains of the sequence, respectively.

The iterator is also polymorphic in the element type being iterated over. Finally, a function acting as the iterator constructor accepts the list to iterate over as an argument. This list is stored as the reference cur, which is part of the $owned$ domain. The implementation of $hasNext$ and $next$ is straightforward, assuming the existence of functions $isCons$, hd, and tl for accessing lists. The $next$ function not only returns the current element, but also advances the cur pointer to refer to the next cell in the list.

4. Formal Semantics

We formulate the semantics of System F_{own} as a set of small-step reduction rules together with syntax-directed typing rules. A number of auxiliary judgment forms are used to reason about type and domain equality, access permissions between domains, and sub-kinding. We first present the dynamic semantics, then the typing rules before considering type soundness.

4.1 Dynamic Semantics

Figure 7 shows the value and intermediate computation forms for System F_{own}, and Figure 8 shows the reduction rules.

During execution, we keep track of the current domain δ, representing the domain that is responsible for the currently executing code.

The reduction relation is of the form $\delta \vdash \mu; t \rightsquigarrow \mu'; t'$. This is read as "In domain δ, the store μ and the expression t reduce in one step to a new store μ' and new expression t'."

We represent reference values with locations l, and index the store μ with locations. As the program reduces, new domains can be created by "letdomain" expressions, and we ensure they are distinct by using an alpha-conversion convention.

Furthermore, we distinguish between lambda expressions in the source program and lambda values by marking values with an overbar, in order to create a distinction between the ability to create a function and the ability to invoke it. As discussed in section 2, the typing rule for a source level function requires that the current domain include a permission to *create* new functions in the domain δ of the function. Once a function $\lambda_\delta x{:}\tau.\ t$ reduces to $\overline{\lambda}_\delta x{:}\tau.\ t$, we no longer make this check; instead, we check (at function application) that the current domain has a permission to access δ.

$$\frac{}{\delta \vdash \mu; \lambda_\gamma x{:}\tau.\ t \rightsquigarrow \mu; \overline{\lambda}_\gamma x{:}\tau.\ t}\ EFun$$

$$\frac{}{\delta \vdash \mu; (\overline{\lambda}_\gamma x{:}\tau.\ t)\ v \rightsquigarrow \mu; ([v/x]t)@\gamma}\ EAppR$$

$$\frac{\gamma \vdash \mu; t \rightsquigarrow \mu'; t'}{\delta \vdash \mu; t@\gamma \rightsquigarrow \mu'; t'@\gamma}\ EAtC$$

$$\frac{}{\delta \vdash \mu; v@\gamma \rightsquigarrow \mu; v}\ EAtR$$

$$\frac{\mu' = \mu, l : v \quad l \notin \text{domain}(\mu)}{\delta \vdash \mu; \text{ref}\ \gamma\ v \rightsquigarrow \mu'; l}\ ERefR$$

$$\frac{}{\delta \vdash \mu; !l \rightsquigarrow \mu; \mu(l)}\ EDeefR$$

$$\frac{\mu' = [\mu | l = v]}{\delta \vdash \mu; l := v \rightsquigarrow \mu'; ()}\ EAssignR$$

$$\frac{}{\delta \vdash \mu; (\overline{\Lambda}_\gamma \alpha{:}K.\ t)\ [\tau] \rightsquigarrow \mu; ([\tau/\alpha]t)@\gamma}\ ETAppR$$

$$\frac{}{\delta \vdash \mu; \Lambda_z \alpha{:}K.\ t \rightsquigarrow \mu; \overline{\Lambda}_z \alpha{:}K.\ t}\ EForall$$

$$\frac{v' = \overline{\text{pack}}\ (\sigma, v)\ \text{as}\ \exists_\gamma \alpha{:}K.\ \tau}{\delta \vdash \mu; \text{pack}\ (\sigma, v)\ \text{as}\ \exists_\gamma \alpha{:}K.\ \tau \rightsquigarrow \mu; v'}\ EPackR$$

$$\frac{v' = \text{pack}\ (\sigma, v)\ \text{as}\ \exists_\gamma \alpha{:}K.\ \tau}{\delta \vdash \mu; \text{unpack}\ (\alpha, x) = v'\ \text{in}\ t_2 \rightsquigarrow \mu; [v/x][\sigma/\alpha]t_2}\ EUnpackR$$

$$\frac{}{\delta \vdash \mu; \text{letdomain}\ z : K\ \text{in}\ t \rightsquigarrow \mu; t}\ EDmIn$$

$$\frac{}{\delta \vdash \mu; \text{letdomain}\ z : K\ \text{into}\ t \rightsquigarrow \mu; t@z}\ EDmInto$$

$$\frac{\delta \vdash \mu; t \rightsquigarrow \mu'; t'}{\delta \vdash \mu; C[t] \rightsquigarrow \mu'; C[t']}\ EContext$$

Figure 8. Dynamic Semantics

Expressions are also extended with locative expressions $t@\gamma$, which represents an expression evaluating in the context of a domain γ. We use this locative in order to identify code a domain other than the current domain – for example, when we perform a function application $(\overline{\lambda}_m x{:}\tau.\ t)\ v$ using the $EAppR$ rule, we substitute the formal parameter with a value, and then mark the substituted expression $[v/x]t$ with the domain m. The locative marker lets us distinguish the body of the function from the context it evaluates in, which greatly simplifies the soundness proof.

Informally, locatives are related to stack inspection, in that they change a principal for the lifetime of a dynamic ccall. However, our locatives could be eliminated in a real implementation (as an examination of the $EAtR$ rule should demonstrate), because have no computational impact – there is no way for a program to test which domain it is in. In addition to function applications, both forms of type application and the "letdomain $z : K$ into t" expression create locatives (in rules $ETApp1$, $ETApp2$, and $EDmInto$, respectively).

Finally, we have two letdomain constructs, which we use to introduce new domains. The only difference between them is that the "letdomain $z : K$ into t" evaluates its

$$\frac{}{\Gamma, z : \text{domain}(P, _ \Rightarrow \delta, P') \vdash z \Rightarrow \delta}\ DCreator$$

$$\frac{}{\Gamma, z : \text{domain}(P, \delta \Rightarrow _, P') \vdash \delta \Rightarrow z}\ DCreated$$

$$\frac{}{\Gamma, z : \text{domain}(P, _ \Rightarrow _, P') \vdash z \Rightarrow z}\ DCrRefl$$

$$\frac{}{\Gamma, z : \text{domain}(P, _ \rightarrow \delta, P') \vdash z \rightarrow \delta}\ DAccessor$$

$$\frac{}{\Gamma, z : \text{domain}(P, \delta \rightarrow _, P') \vdash \delta \rightarrow z}\ DAccessed$$

$$\frac{}{\Gamma, z : \text{domain}(P, _ \rightarrow _, P') \vdash z \rightarrow z}\ DAcRefl$$

Figure 9. Domain Definition, Creation, and Access Rules

subexpression t in the new domain z it just created, and "letdomain $z : K$ in t" evaluates t in the current domain. It is worth noting that repeated evaluations of a letdomain expression (for example, if it is part of a function body) will produce *different* domains. We enforce this by assuming an alpha-conversion convention similar to the one for variable bindings.

The rules for reference creation, assignment, and dereference are completely standard. Finally, the congruence rule $EContext$ allows reduction to proceed within an evaluation context.

4.2 Domain Access Rules

We begin with the most interesting component of the static semantics: the rules showing when a domain has permission to create or access references within a domain.

We have two judgments: first, $\Gamma \vdash \delta \Rightarrow \gamma$, determines whether domain δ has permission to *create* references or code in domain γ, and second, $\Gamma \vdash \delta \rightarrow \gamma$, determines whether domain δ can *access* objects in domain γ.[1] We can read either judgment as "given a domain heap $\mathcal{D}$ and a variable context Γ, domain δ has the ability to touch domain γ".

The first rule, $DCreator$, establishes that a domain z can can create functions and references in another domain, and conversely, $DCreated$ grants another domain permission to create objects in z. The $DCrSelf$ rules is used to determine whether a particular domain is reflexive – whether it can create objects in itself. There are a parallel set of rules for domain access, with $DAccessor$ granting z permission to access another domain, $DAccessed$ granting another domain the right to access z, and $DAcRefl$ granting z reflexive access.

It's worth observing once again that the access rules are not transitive. However, if domain γ has both create and access permission to domain δ, then it can access anything δ can access simply by creating a new function in δ and then calling it. Thus, create permission effectively enables one domain to impersonate another, and so it should be used with care.

4.3 Typing Rules

Figures 10 and 11 show the typing rules for System F_{own}. The typing relation is of the form $\gamma; \Sigma; \Gamma \vdash e : \tau$, read, "In the

[1] Recall that to "access" γ means to read or write pointers from γ, or to invoke functions or to open packages in it.

$$\frac{x:\tau \in \Gamma}{\delta;\Sigma;\Gamma \vdash x:\tau}\ Varref$$

$$\frac{\gamma;\Sigma;\Gamma, x:\tau \vdash t:\tau' \quad \Gamma \vdash \delta \Rightarrow \gamma}{\delta;\Sigma;\Gamma \vdash \lambda_\gamma x{:}\tau.\ t:\tau \rightarrow_\gamma \tau'}\ Lambda$$

$$\frac{\gamma;\Sigma;\Gamma, x:\tau \vdash t:\tau'}{\delta;\Sigma;\Gamma \vdash \overline{\lambda}_\gamma x{:}\tau.\ t:\tau \rightarrow_\gamma \tau'}\ LambdaVal$$

$$\frac{\begin{array}{ll}\delta;\Sigma;\Gamma \vdash t_1:\tau \rightarrow_\gamma \tau' & \\ \delta;\Sigma;\Gamma \vdash t_2:\tau & \Gamma \vdash \delta \rightarrow \gamma\end{array}}{\delta;\Sigma;\Gamma \vdash t_1\ t_2:\tau'}\ App$$

$$\frac{\gamma;\Sigma;\Gamma \vdash t_1:\tau \quad \Gamma \vdash \delta \rightarrow \gamma}{\delta;\Sigma;\Gamma \vdash t_1@\gamma:\tau}\ At$$

$$\frac{l:\text{ref}\ \gamma\ \tau \in \Sigma}{\delta;\Sigma;\Gamma \vdash l:\text{ref}\ \gamma\ \tau}\ Label$$

$$\frac{\delta;\Sigma;\Gamma \vdash t:\tau \quad \Gamma \vdash \delta \Rightarrow \gamma}{\delta;\Sigma;\Gamma \vdash \text{ref}\ \gamma\ t:\text{ref}\ \gamma\ \tau}\ Ref$$

$$\frac{\delta;\Sigma;\Gamma \vdash t:\text{ref}\ \gamma\ \tau \quad \Gamma \vdash \delta \rightarrow \gamma}{\delta;\Sigma;\Gamma \vdash !t:\tau}\ DeRef$$

$$\frac{\begin{array}{ll}\delta;\Sigma;\Gamma \vdash t_1:\text{ref}\ \gamma\ \tau & \\ \delta;\Sigma;\Gamma \vdash t_2:\tau & \Gamma \vdash \delta \rightarrow \gamma\end{array}}{\delta;\Sigma;\Gamma \vdash t_1 := t_2:\text{unit}}\ Assign$$

$$\frac{\gamma;\Sigma;\Gamma, z:K \vdash t:\tau \quad \Gamma \vdash \delta \Rightarrow \gamma \quad \Gamma \vdash K:\text{kind}}{\delta;\Sigma;\Gamma \vdash \Lambda_\gamma z{:}K.\ t:\forall_\gamma z{:}K.\ \tau}\ \Lambda_1$$

$$\frac{\gamma;\Sigma;\Gamma, z:K \vdash t:\tau \quad \Gamma \vdash K:\text{kind}}{\delta;\Sigma;\Gamma \vdash \overline{\Lambda}_\gamma z{:}K.\ t:\forall_\gamma z{:}K.\ \tau}\ \overline{\Lambda}_1$$

$$\frac{z;\Sigma;\Gamma, z:\text{domain}(P) \vdash t:\tau \quad fv(P)=\emptyset}{\delta;\Sigma;\Gamma \vdash \Lambda z{:}\text{domain}(P).\ t:\forall z{:}\text{domain}(P).\ \tau}\ \Lambda_2$$

$$\frac{z;\Sigma;\Gamma, z:\text{domain}(P) \vdash t:\tau \quad fv(P)=\emptyset}{\delta;\Sigma;\Gamma \vdash \overline{\Lambda} z{:}\text{domain}(P).\ t:\forall z{:}\text{domain}(P).\ \tau}\ \overline{\Lambda}_2$$

$$\frac{\begin{array}{ll}\delta;\Sigma;\Gamma \vdash t:\forall_\gamma \alpha{:}K.\ \tau & \Gamma \vdash K' \leq K \\ \delta;\Sigma;\Gamma \vdash \omega:K' & \Gamma \vdash \delta \rightarrow \gamma\end{array}}{\delta;\Sigma;\Gamma \vdash t[\omega]:[\omega/\alpha]\tau}\ TApp1$$

$$\frac{\begin{array}{l}\delta;\Sigma;\Gamma \vdash t:\forall \alpha{:}\text{domain}(P).\ \tau \\ \delta;\Sigma;\Gamma \vdash \gamma:K \\ \Gamma \vdash K \leq \text{domain}() \\ \Gamma \vdash \delta \rightarrow \gamma \text{ if } _ \rightarrow _ \in P \\ \Gamma \vdash \delta \Rightarrow \gamma \text{ if } _ \Rightarrow _ \in P\end{array}}{\delta;\Sigma;\Gamma \vdash t[\gamma]:[\gamma/\alpha]\tau}\ TApp2$$

Figure 10. Static Semantics

context of domain γ, store type Σ and type environment Γ, expression e has type τ."

Most of the typing rules are fairly straightforward, with the main differences between System F_{own} and the standard typing rules lying in the addition of checks to verify access permissions. For example, notice that $Lambda$ checks for a creation permission, and that $LambdaVal$ does not, as discussed earlier, and that the Λ_1 and $\exists I$ rules both have corresponding creation permission checks and value forms. The Ref rule is standard, with an extra check for creation rights, and the App, $DeRef$ and $Assign$ rules are the familiar rules for function application, type application, dereference and assignment, with an additional check for access permissions. (The Λ_1 and related rules also check for the well-formedness of types and domains. There is nothing unexpected in this judgment, and we elide it for reasons of space.)

The $TApp1$ rule also follows this pattern, with the additional feature that it permits a subkind of its expected argument. The subkinding relation is defined in Figure 13; essentially, we define a domain δ to be a subkind of γ if it offers at least the permissions of γ. This lets us pass domains with more permissions as arguments to functions that want a domain with some set of permissions.

The typing rule for Λ_2 differs from this pattern, because it checks its body in an environment in which its domain is *changed* to its argument, a domain about which no assumptions about any other domains are made – the condition $fv(P) = \emptyset$ means that only the permissions $_ \Rightarrow _$ and $_ \rightarrow _$ can be in P. If the body typechecks with this minimal assumption, then it can safely be run in any domain with the permissions P. Thus, the application rule $TApp2$ also varies from the established pattern. We require that the argument passed to it be one that the current domain have permissions for, because the body of a Λ_2 can, given a rich enough P, create references or access pointers in the domain it was given as an argument. Consider "$\Lambda\alpha{:}\text{domain}(_ \Rightarrow _).\ \text{ref}\ \alpha\ ()$" – which has the type "$\forall\alpha{:}\text{domain}(_ \Rightarrow _).\ \text{ref}\ \alpha\ \text{unit}$". If the current domain did not have permission to create references in the domain it was given as an argument, access protection could be violated.

Finally, we have the $DmIn$ and $DmInto$ rules. Both of these create a new domain and typecheck the body with the assumption that this domain is available, varying only in the question of which domain the body is checked in. Both also check that their domain z is creatable, which is a check that z is given no permissions that the current domain does not have. This judgment is in Fig 12.

One final question remains: the typing and reduction judgments require a domain, but how can we get such a domain before the program starts evaluating? Our solution is to begin evaluation in a context with initial domain $\Gamma = \delta : \text{domain}(_ \Rightarrow _, _ \rightarrow _)$, and we typecheck against this initial domain.

Figure 13 shows the main rules for subkinding in our system. The type kind is a subkind of itself, and two domains $\text{domain}(P)$ and $\text{domain}(Q)$ are in a subkind relationship if $Q \subseteq P$ (i.e., if P grants at least the permissions of Q). We retain this judgment in order to show that it is possible to actually invoke our domain-universal form.

4.4 Soundness

We prove the soundness of System F_{own} with the standard combination of progress and type preservation theorems.

Theorem 1 (Progress)
If $\delta;\Sigma;\epsilon \vdash t : \tau$ and $\Sigma \vdash \mu$, then either t is a value or $\delta \vdash \mu; t \rightsquigarrow \mu'; t'$.

Proof: We prove progress with a conventional induction over the structure of the typing derivation. The progress proof is wholly conventional; all of the interesting structure

$$\frac{\gamma;\Sigma;\Gamma\vdash t:[\sigma/x]\tau \quad \Gamma\vdash\delta\Rightarrow\gamma}{\delta;\Sigma;\Gamma\vdash \text{pack}\ (\sigma,t)\ \text{as}\ \exists_\gamma x{:}K.\ \tau : \exists_\gamma x{:}K.\ \tau}\ \exists I$$

$$\frac{\gamma;\Sigma;\Gamma\vdash v:[\sigma/x]\tau}{\delta;\Sigma;\Gamma\vdash \overline{\text{pack}}\ (\sigma,v)\ \text{as}\ \exists_\gamma x{:}K.\ \tau : \exists_\gamma x{:}K.\ \tau}\ \overline{\exists I}$$

$$\frac{\begin{array}{l}\delta;\Sigma;\Gamma\vdash t:\exists_\gamma x{:}K.\ \tau\\ \delta;\Sigma;\Gamma,\alpha:K,x:\tau\vdash t':\sigma \quad \Gamma\vdash\delta\to\gamma\end{array}}{\delta;\Sigma;\Gamma\vdash \text{unpack}\ (\alpha,x)=t\ \text{in}\ t':\sigma}\ \exists E$$

$$\frac{\delta;\Sigma;\Gamma,z:K\vdash t:\tau \quad \delta;\Gamma\vdash K\ \text{creatable}}{\delta;\Sigma;\Gamma\vdash \text{letdomain}\ z:K\ \text{in}\ t:\tau}\ DmIn$$

$$\frac{\gamma;\Sigma;\Gamma,z:K\vdash t:\tau \quad \delta;\Gamma\vdash K\ \text{creatable}}{\delta;\Sigma;\Gamma\vdash \text{letdomain}\ z:K\ \text{in}\ t:\tau}\ DmInto$$

Figure 11. More Static Semantics

$$\frac{\delta;\Gamma\vdash P\ \text{accessible}}{\delta;\Gamma\vdash \text{domain}(P)\ \text{creatable}}\ Base$$

$$\frac{}{\delta;\Gamma\vdash\epsilon\ \text{accessible}}\ AccessEmpty$$

$$\frac{\Gamma\vdash\delta\rightsquigarrow\gamma \quad \rightsquigarrow\in\{\to,\Rightarrow\} \quad \delta;\Gamma\vdash P\ \text{accessible}}{\delta;\Gamma\vdash P,_\rightsquigarrow\gamma\ \text{accessible}}\ AccessFrom$$

$$\frac{\Gamma\vdash\gamma\rightsquigarrow\delta \quad \rightsquigarrow\in\{\to,\Rightarrow\} \quad \delta;\Gamma\vdash P\ \text{accessible}}{\delta;\Gamma\vdash P,\gamma\rightsquigarrow_\ \text{accessible}}\ AccessTo$$

$$\frac{\Gamma\vdash\delta\rightsquigarrow\delta \quad \rightsquigarrow\in\{\to,\Rightarrow\} \quad \delta;\Gamma\vdash P\ \text{accessible}}{\delta;\Gamma\vdash P,_\rightsquigarrow_\ \text{accessible}}\ AccessRefl$$

Figure 12. Creatability

$$\frac{}{\Gamma\vdash \text{type}\le\text{type}}\ SType$$

$$\frac{\Gamma\vdash P\le P'}{\Gamma\vdash \text{domain}(P)\le\text{domain}(P')}\ SDomain$$

$$\frac{}{\Gamma\vdash\epsilon\le\epsilon}\ SEmpty$$

$$\frac{\Gamma\vdash P,\phi,Q\le P'}{\Gamma\vdash P,Q\le P'}\ SShrink$$

Figure 13. Subtyping

is in the type preservation theorem. ■

Theorem 2 (Type Preservation)
If $\delta;\Sigma;\Gamma_\delta\vdash t:\tau$, and $\Sigma\vdash\mu$, and $\delta\vdash\mu;t\rightsquigarrow\mu';e'$, then there exists $\Sigma'\supseteq\Sigma$ and $\Gamma'_\delta\supseteq\Gamma_\delta$ such that $\delta;\Sigma';\Gamma'\vdash t':\tau$ and $\Sigma'\vdash\mu'$.

Proof: We do this proof by an induction on the derivation relation. The unusual features of this proof, relative to the soundness proof of System F, are that first, we need to prove a *domain value* lemma to show that a closed, well-typed value is well-typed at any domain, and that the substitution lemma only holds when substituting values for variables, which means that the soundness proof for System F_{own} ends up relying critically on the language being call-by-value.

Secondly, our type preservation lemma allows Γ_δ to grow as the program evaluates. However, this context only contains domains, which have no term representation, so none of the problems of evaluating open terms ever arise. (If desired, a separate context for domains could be created.) ■

Once we have a soundness proof, we can use it prove that the access permissions are always respected:

Theorem 3 (Access Correctness)
If there is a derivation $\mathcal{C}=\delta;\Sigma;\Gamma_\delta\vdash\mathcal{E}[t]:\tau$, and if

- *$t=!l$ (with $\Sigma(l)=$ ref $\gamma\ \sigma$),*
- *$t=l:=v$ (with $\Sigma(l)=$ ref $\gamma\ \sigma$),*
- *$t=(\overline{\lambda}_\gamma x{:}\tau.\ t)v$,*
- *$t=(\overline{\Lambda}_\gamma\alpha{:}K.\ t')[\omega]$,*
- *or $t=$ unpack $(\alpha,x)=v$ in t' (with $v=\overline{\text{pack}}\ (\omega,v)$ as $\exists_\gamma\alpha{:}K.\ \tau$),*

and $\delta\vdash\mu;\mathcal{E}[t]\rightsquigarrow\mu';\mathcal{E}[t']$, then there exists a domain ω such that there is a derivation $\omega;\Sigma;\Gamma'_\delta;t:\sigma$ which is a subderivation of $\mathcal{C}$, and $\Gamma'_\delta\vdash\omega\to\gamma$.

Here, $\mathcal{E}$ represent contexts that can go inside locatives ($\mathcal{E}$ ::= ... | $\mathcal{E}@m'$) unlike the evaluation contexts defined earlier.

5. Related Work

Ownership. A number of early research projects, including Islands [17] and Balloons [3], provided a way to encapsulate one object within another. The term “ownership” is due to the Flexible Alias Protection project [22, 10], which added ownership parameters in order to support object-oriented idioms like collection classes. Early ownership systems all enforced a strong encapsulation property known as owners-as-dominators: an owned object is dominated by its owner, in the sense that all paths in the heap from external objects to the owned object must go through its owner. In practice, this property is too restrictive: idioms such as object-oriented iterators, callback functions, and calls to the standard library all may require more flexible heap structures. A number of solutions have been proposed for this problem, including supporting stack-based aliases to owned objects [9], capability-based encapsulation [2], or allowing inner objects to have privileged access to the state of their enclosing objects [8, 6].

Several systems build on the owners-as-dominators property to provide secondary properties including safe concurrency [6], safe memory management [7], reasoning about effects [9], and abstraction [4]. While the owners-as-dominators property loses its meaning in System F_{own} (which is not object-oriented), we believe that our system in principle can enable similar kinds of reasoning–an important direction for future work.

Clarke et al.'s Simple Ownership Types system introduced the idea of *ownership contexts*, which provided additional flexibility by decoupling ownership from individual objects, and studied ownership in a foundational object calculus [11, 8]. Our previous work on Ownership Do-

mains builds on ownership contexts by introducing ownership domains with user-defined access permissions, but creation permissions and a hierarchy of domains were still hardwired [1].

Our system is simpler, more uniform, and more flexible than our previous work, because we have added explicit creation permissions, done away with the idea of domain hierarchy, and permit the creation of new domains at will. As a result, the previous systems can be encoded in System F_{own}. Furthermore, unlike the previous work, we work in an imperative version of System F, rather than in an object calculus. This lets us work in a simpler and better-understood system, and opens the door to using ownership in functional and procedural languages.

Region and Effect Systems. Region-based memory management systems group references into ownership-domain-like regions [28, 7]. Our system differs from region systems in several ways. First of all, in order to accurately track access permissions, our domains include not only references but also functions and existential packages.

Second, region systems are intended to support statically checkable explicit memory management, requiring these systems to restrict inter-region references according to a stack-like discipline or linear logic. Our system's goal of enforcing encapsulation allows us to support more general aliasing patterns, according to the access policies specified by the programmer.

Type and effect systems[27] are another way of tracking access to state. The main difference is that such systems are usually track the transitive closure all possible effects, and while System F_{own} tracks access, which is not a transitive property. Thus, while effect systems verify that a function only affects a given part of the program's state, System F_{own} verifies that any effects a function has are mediated through the proper information-hiding interface. Tracking only local access also allows our annotations to remain small compared to a system that tracks transitive effects.

Other Related Work. Grossman et al.'s syntactic type abstraction is similar to our work in tagging data with labels that represent the principals that own the data [16]. The domains of our ownership system resemble the principals, except that new domains can arise dynamically during program execution, rather than partitioning the program statically.

Furthermore, our notion of locatives bears a strong resemblance to the work on stack inspection [14]. We use locatives of the form $e@\delta$ in order to identify the bodies of function invocations and keep track of their permissions. This closely resembles the dynamic grant of permissions in a stack-inspection calculus. System F_{own} differs from stack inspection in that it supports an unbounded number of principals, each of which protects its own data from other principals according to a well-defined access policy.

Adoption is another mechanism, similar to ownership, that can be used to encapsulate one object within another [12]. Although adoption can enforce strong encapsulation in the presence of state, it is less flexible than ownership in the programming idioms that it allows–for example, iterator and callback objects would be forbidden in this approach.

Confined types [5] restrict aliases of an object to within a particular package, a weaker but more lightweight notion compared to the object-based encapsulation provided by ownership domains.

Systems like alias types [29] and separation logic [26] provide a finer control of aliasing compared to ownership domains, but are also much more intricate, requiring more declarations for the same level of reasoning about aliasing. As future work, we plan on encoding ownership domains in separation logic, to more precisely characterize how they modularize the use of state [24].

Leino et al.'s data groups [19] and Greenhouse et al.'s regions [15] are similar to ownership domains. Here groups and regions refer to sets of fields rather than sets of objects, references, and functions, and are used to reason about effects rather than aliasing. Banerjee and Naumann [4] make use of this methodology to prove an abstraction theorem for a specification-based ownership technique.

The functional language Haskell has the concept of a *state monad*, which is a type representing computations with state. This idea has strong connections to region systems, which can be regarded [13] as a way of creating indexed families of state monads. We speculate that future work could relate the two via a mutual connection to modal logic. Monads give rise to the diamond modality in constructive modal logic. Our permissions form a globally-visible access control list, as in information flow analysis, and so it may be possible to model this with a box modality as in [21].

6. Conclusion

This paper has shown how an ownership type system can be integrated with the type theory of System F, including abstract data types, first-class functions, and universal and existential quantification. The resulting system is simpler and more flexible than previous ownership type systems, yet provides stronger encapsulation guarantees.

In future work, we plan to integrate this design into a user-level language that will combine the benefits of ownership and advanced language constructs. We are currently working on supporting local type inference for ownership in the setting of object-oriented languages. Finally, we would like to investigate connections between ownership and logic.

7. Acknowledgments

Thanks to John Boyland, David Clarke, Karl Crary, Aleks Nanevski, Kevin Donnelly, Joshua Dunfield, Dan Grossman, Robert Harper, Donna Malayeri, Kevin Watkins, and the anonymous reviewers for their comments on improving this paper. This work was supported in part by the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298 and National Science Foundation Grants CCR-0204242 and CCR-0204047.

References

[1] J. Aldrich and C. Chambers. Ownership Domains: Separating Aliasing Policy from Mechanism. In *European Conference on Object-Oriented Programming*, June 2004.

[2] J. Aldrich, V. Kostadinov, and C. Chambers. Alias Annotations for Program Understanding. In *Object-Oriented Programming Systems, Languages, and Applications*, November 2002.

[3] P. S. Almeida. Balloon Types: Controlling Sharing of State in Data Types. In *European Conference on Object-Oriented Programming*, June 1997.

[4] A. Banerjee and D. A. Naumann. Representation Independence, Confinement, and Access Control. In *Principles of Programming Languages*, January 2002.

[5] B. Bokowski and J. Vitek. Confined Types. In *Object-Oriented Programming Systems, Languages, and Applications*, November 1999.

[6] C. Boyapati, R. Lee, and M. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *Object-Oriented Programming Systems, Languages, and Applications*, November 2002.

[7] C. Boyapati, A. Salcianu, J. William Beebee, and M. Rinard. Ownership Types for Safe Region-Based Memory Mangement in Real-Time Java. In *Programming Language Design and Implementation*, June 2003.

[8] D. Clarke. *Object Ownership & Containment*. PhD thesis, University of New South Wales, July 2001.

[9] D. Clarke and S. Drossopoulou. Ownership, Encapsulation, and the Disjointness of Type and Effect. In *Object-Oriented Programming Systems, Languages, and Applications*, November 2002.

[10] D. G. Clarke, J. M. Potter, and J. Noble. Ownership Types for Flexible Alias Protection. In *Object-Oriented Programming Systems, Languages, and Applications*, October 1998.

[11] D. G. Clarke, J. M. Potter, and J. Noble. Simple Ownership Types for Object Containment. In *European Conference on Object-Oriented Programming*, 2001.

[12] M. Fahndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *Programming Language Design and Implementation*, pages 13–24. ACM Press, 2002.

[13] M. Fluet and G. Morisett. Monadic regions. In *International Conference on Functional Programming*, 2004.

[14] C. Fournet and A. D. Gordon. Stack inspection: theory and variants. In *Symposium on Principles of Programming Languages*, pages 307–318, 2002.

[15] A. Greenhouse and J. Boyland. An Object-Oriented Effects System. In *European Conference on Object-Oriented Programming*, June 1999.

[16] D. Grossman, G. Morrisett, and S. Zdancewic. Syntactic Type Abstraction. *Transactions on Programming Languages and Systems*, 22(6):1037–1080, November 2000.

[17] J. Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *Object-Oriented Programming Systems, Languages, and Applications*, October 1991.

[18] Y. L. J-Y Girard and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[19] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using Data Groups to Specify and Check Side Effects. In *Programming Language Design and Implementation*, June 2002.

[20] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. *Transactions on Programming Languages and Systems*, 10(3):470–502, 1988.

[21] K. Miyamoto and A. Igarashi. A modal foundation for secure information flow. In *Workshop on Foundations of Computer Security*, pages 187–203, 2004.

[22] J. Noble, J. Vitek, and J. Potter. Flexible Alias Protection. In *European Conference on Object-Oriented Programming*, 1998.

[23] M. Odersky. Programming in Scala. Book draft available at http://scala.epfl.ch/, 2004.

[24] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Principles of Programming Languages*, 2004.

[25] D. L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

[26] J. C. Reynolds. Separation Logic: a Logic for Shared Mutable Data Structures. In *Logic in Computer Science*, July 2002.

[27] J.-P. Talpin and P. Jouvelot. The type and effect discipline. In *Information and Computation*, pages 245–296, 1994.

[28] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109–176, 1997.

[29] D. Walker and G. Morrisett. Alias Types for Recursive Data Structures. In *International Workshop on Types in Compilation*, September 2000.